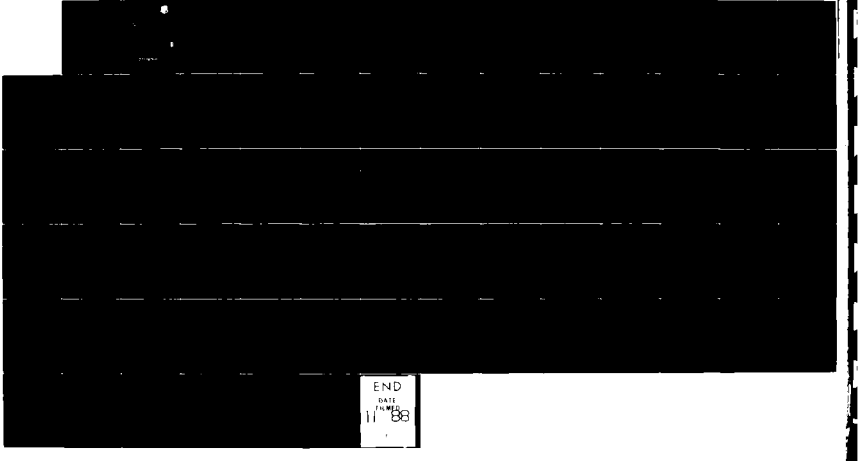
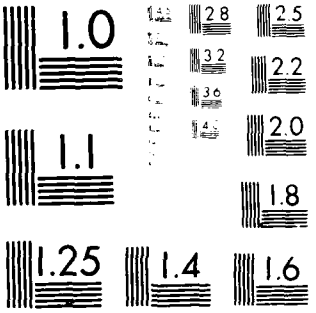
AD-A199 425 A MATHEMATICAL THEORY OF ASYMPTOTIC COMPUTATION(U) 1/1
ODYSSEY RESEARCH ASSOCIATES INC ITHACA NY D SUTHERLAND
DEC 87 RADC-TR-87-261 F30602-86-C-0116

UNCLASSIFIED F/G 12/2 NL



END
DATE FILMED
11 88



(4)

AD-A199 425

RADC-TR-87-261
Final Technical Report
December 1987

A MATHEMATICAL THEORY OF ASYMPTOTIC COMPUTATION

Odyssey Research Associates

Sponsored by
Strategic Defense Initiative Office

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-261 has been reviewed and is approved for publication.

APPROVED: Donald M. Elefante

DONALD M. ELEFANTE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: James W. Hyde III

JAMES W. HYDE, III.
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

A MATHEMATICAL THEORY OF ASYMPTOTIC COMPUTATION

David Sutherland

Contractor: Odyssey Research Associates
Contract Number: F30602-86-C-0116
Effective Date of Contract: 1 May 86
Contract Expiration Date: 30 Apr 89
Short Title of Work: Formal Verification of SDI Mathematical Software
Period of Work Covered: May 86 - Oct 87

Principal Investigator: David Sutherland
Phone: (607) 277-2020

Project Engineer: Donald M. Elefante
Phone: (315) 330-3241

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Donald M. Elefante (COTC), Griffiss AFB NY, 13441-5700, under Contract F30602-86-C-0116.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-261
6a. NAME OF PERFORMING ORGANIZATION Odyssey Research Associates 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)
6c. ADDRESS (City, State, and ZIP Code) 301A Harris B. Dates Drive Ithaca NY 14850-1313	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Strategic Defense Initiative Office 8b. OFFICE SYMBOL (If applicable) S-BM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-86-C-0116
8c. ADDRESS (City, State, and ZIP Code) Office of the Secretary of Defense Wash DC 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 63223C; PROJECT NO.: B413; TASK NO: 03; WORK UNIT ACCESSION NO.: 03
11. TITLE (Include Security Classification) A MATHEMATICAL THEORY OF ASYMPTOTIC COMPUTATION	
12. PERSONAL AUTHOR(S) David Sutherland	
13a. TYPE OF REPORT Final	13b. TIME COVERED FROM May 86 TO Oct 87
14. DATE OF REPORT (Year, Month, Day) December 1987	15. PAGE COUNT 80
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Formal Verification; Verification Over The Reals; Verification of Mathematical Programs
(See Reverse)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

One of the major problems encountered in trying to formally verify the correctness of computer programs that use real arithmetic (hereinafter referred to as "mathematical programs") is that the mathematical properties of real arithmetic operations in computers are much more complicated and much harder to work with than the mathematical properties of the corresponding ideal mathematical operations. This occurs because the real number type implemented on a finite computer is not the same as the ideal, mathematical real number type. A finite machine can only represent finitely many different real numbers, whereas there are infinitely many ideal real numbers.

The idea behind the theory of asymptotic computing is to develop techniques to prove that the accuracy of a mathematical program goes to infinity (e.g., larger and larger numbers of representation bits for mantissas and exponents used in binary floating point arithmetic).

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Donald M. Elefante	22b. TELEPHONE (Include Area Code) (315) 330-3241 22c. OFFICE SYMBOL RADC (COTC)

DD Form 1473, JUN 86 Previous editions are obsolete

> The theory of asymptotic computing, then, is essentially a general formalization of the notions of "accuracy" and "accuracy going to infinity", but without having to show how fast convergence happens (a major source of difficulty in numerical analysis).

18. SUBJECT TERMS (Continued)

Formal Verification Theory
Reals Verification
Software Verification

Contents

Appendix A

Chapter 1

Introduction

We wish to formally verify the correctness of computer programs that use real arithmetic (hereinafter referred to as "mathematical programs"). The real number type implemented on a finite computer is not the same as the ideal, mathematical real number type because a finite machine can only represent finitely many different real numbers, whereas there are infinitely many ideal real numbers.

One of the major problems encountered in trying to verify mathematical programs is that the mathematical properties of real arithmetic operations in computers are much more complicated and much harder to work with than the mathematical properties of the corresponding ideal mathematical operations. For example, ideal real addition is associative; floating point real addition is not. How can we handle this difficulty?

One way that might come to mind is to "pretend" that the machine reals are the same as the ideal reals. Strictly speaking, this is not true. However, this is what is done for programs which use integer arithmetic. Why is it OK for integer programs? If we verify a statement about an integer program like "on any input n, the program will terminate and return n^2" based on the assumption that the machine integers are the same as the ideal integers, we will actually have established that on any input n that is representable in the machine the program is running on, the program will either cause an

overflow or will terminate and return n^2. This is because integer arithmetic in finite machines is identical to ideal integer arithmetic *when overflow does not occur*. Unfortunately, the same is not true of machine real arithmetic. Machine real arithmetic can also deviate from ideal arithmetic by underflow or roundoff. Thus, if we verify a statement like "on any input $x \neq 0$, the program will terminate and return $1/x$" based on the assumption that the machine reals are the same as the ideal reals, we will actually have established that on any input x that is representable in the machine the program is running on, the program will terminate and return $1/x$ *if* no overflow, underflow or roundoff occurs. Since roundoff occurs much more frequently in real arithmetic than overflow occurs in integer arithmetic, we have established a much weaker statement. In fact, the statement we have actually established is so weak that it is useless. Thus, whatever axioms we assume about machine real numbers, our assumptions must recognize at least some of the differences between machine reals and ideal reals.

We could formulate a collection of axioms which are satisfied by all implementations of real numbers on finite machines, or at least all implementations in a certain general class, like machines that use binary floating point arithmetic. Such an axiom system would have to incorporate some unspecified constants (e.g. the number of bits of mantissa and exponent in the case of binary floating point arithmetic) in order to be valid on machines of various sizes. One could then verify properties like "on any input x representable on the machine, the program will terminate and return the square root of x correct to t decimal places" where t would be some expression involving the unspecified constants. Having done such a verification, given a machine, we could determine the values of the constants for that particular machine, and get a lower bound on the number of decimal places of accuracy (by plugging the values of the constants into t and evaluating it). This is the kind of verification one would really *like* to do, but it is very difficult. The difficulty comes primarily from the fact that one must perform complicated numerical analyses to get such a hard bound on the number of decimal places of accuracy.

What we have attempted to do with the Theory of Asymptotic Computation is to "factor out" the hard numerical details. Let's return for a moment to the case of stating axioms in terms of unspecified constants about the

machine's accuracy. We'd like it to be the case that if we plug in values of these constants corresponding to more and more accurate machines (e.g. larger and larger numbers of bits for the mantissa and exponent in the case of binary floating point arithmetic), the value of the term t goes to ∞. In other words, running the program on more and more accurate machines gives better and better accuracy in the result computed by the program. The idea of the theory of asymptotic computation is to develop techniques to prove *that* the accuracy of the program goes to ∞ as the accuracy of the underlying machine goes to ∞, without having to show *how fast* this convergence happens, which is where most of the messy numerical analysis comes in.

The theory of asymptotic computation is essentially a general formalization of the notions of "accuracy" and of accuracy "going to ∞".

In Chapter 2 we describe the Theory of Asymptotic Computation. This chapter includes:

- the programming language we are using for specifying algorithms
- a semantics for the language
- the definition of what it means for a program to satisfy a certain input/output specification asymptotically.

In Chapter 3 we give a formulation of the Theory in Nonstandard Mathematics. This formulation makes the definitions less complicated and more intuitive.

In Chapter 4 we apply the formulation of Chapter 3 to verify a program to find roots of a real valued function.

Chapter 2

A Mathematical Theory of Asymptotic Computation

2.1 A Motivating Example

We will explain the Theory by first considering a very simple program. We will give a semantics for the program, state what it means for the program to be asymptotically correct, and prove it asymptotically correct. We will then obtain the Theory of Asymptotic Computation as a generalization of this example.

The program we will consider is a program to sum 3 real numbers. The 3 numbers to be summed will be given to the program as the values of 3 variables, A, B and C. The output will be stored in a variable RESULT. Here is the program:

```
10    RESULT := A + B;
20    RESULT := RESULT + C;
30    END;
```

What do we mean by "asymptotic correctness" for this program, and how

would we prove it asymptotically correct? First of all, in order to prove anything about a program we *must represent* it as a mathematical object. At any point during a possible "run" of the program, either:

1. control is at one of the three statements in the program, with some subset of the variables assigned real number values, or
2. some kind of exception (e.g. overflow) has occurred and the program has terminated abnormally.

Thus, the entire history of a run can be expressed by giving the (finite) sequence of "states" the program has passed through, where by "state" we mean a statement number and an assignment of some variables to real numbers, and telling whether an exception has occurred or not. We will find it convenient to incorporate both of these pieces of information into a single finite sequence each of whose entries is either a state or a distinguished element "!" which stands for the occurrence of an exception. We can think of the entries in such a sequence as events, with a state s being thought of as the "event" of going into state s, and ! being thought of as the event of an exception occurring. The events occur in the sequence in the order in which they occurred. We will call such a sequence a *trace* of the program. The collection of all traces which could occur during *any* run of the program defines the semantics of the program's execution. We will call such a collection of traces an *event system* over the set of states. We will represent the program as an event system. Note that such an event system T is always nonempty (it contains at least $\langle\rangle$, the sequence of events which have occurred before anything at all has happened), and it is always closed under initial segment, i.e. $\forall \sigma \in T. \forall \tau$, if $\tau \preceq \sigma$ then $\tau \in T$. Sets of finite sequences having these two properties are called *trees of finite sequences*.

What are the possible traces of the above program? Let's first answer this question for the case in which the machine real number type is exactly the same as the ideal real number type. We will denote a state by an n-tuple consisting of a statement number and a sequence of variable bindings to describe the assignment of variables. A variable binding will just be a variable name followed by an arrow and the value that the variable is bound

to. If a variable does not appear in the list of bindings, it is not assigned a value by the state.

As noted above, the empty sequence, $\langle\rangle$, is a trace. We assume that A, B and C are defined whenever the program is started up, but their values can be anything, and RESULT may or may not be defined. Also, control must initially be at statement 10. Thus, all sequences of the form

$$\langle\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2\rangle\rangle$$

or

$$\langle\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w\rangle\rangle$$

will be traces, and no other sequences of length 1 will be traces. From statement 10 the program must go to statement 20, with the new value of RESULT being the sum of the old values of A and B (values of A, B, and C unchanged). In terms of traces, this means that all sequences of the form

$$\langle\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2\rangle,$$
$$\langle 20, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow x_0 + x_1\rangle\rangle$$

or

$$\langle\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w\rangle,$$
$$\langle 20, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow x_0 + x_1\rangle\rangle$$

will be traces, and no other sequences of length 2 will be traces. Similarly, all sequences of the form

$$\langle\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2\rangle,$$
$$\langle 20, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow x_0 + x_1\rangle,$$
$$\langle 30, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow x_0 + x_1 + x_2\rangle\rangle$$

or

$$\langle\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w\rangle,$$
$$\langle 20, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow x_0 + x_1\rangle,$$
$$\langle 30, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow x_0 + x_1 + x_2\rangle\rangle$$

will be traces, and no other sequences of length 3 will be traces. Since the program halts at statement 30, the set of state sequences will contain no sequences of length > 3. Also, if the machine addition is ideal, no exceptions can occur, so no trace of the ideal system will contain !.

Now we examine what the traces for the program running on a finite machine could look like. First of all, what can we reasonably assume about the traces that will be true on *any* finite machine? We can presumably at least assume the following:

- Control always starts at statement 10 with A, B and C assigned values. Formally, this means that if $\langle s\rangle$ is a trace, then s must either be of the form

  $$\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2\rangle$$

  or

  $$\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w\rangle$$

  Note that we do *not* assume the converse, that for all states s of this form, $\langle s\rangle$ is a trace. This would require that there be infinitely many different states that the program can start in, which is not possible on a finite machine.

- If control is at statement 10, then either an exception will occur, or the program will go to a state in which control is at statement 20 and the values of A, B and C will be unchanged and RESULT will be assigned a value. Formally, this means that if $\sigma \hat{}\langle e, e'\rangle$ is a trace, and e is a state of the form

$$\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2 \rangle$$

or

$$\langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w \rangle$$

then either $e' = !$ or $\langle e, e' \rangle$ is of one of the following two forms:

$$\langle \langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2 \rangle,$$
$$\langle 20, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w' \rangle \rangle$$

$$\langle \langle 10, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w \rangle,$$
$$\langle 20, \mathrm{A} \Rightarrow x_0, \mathrm{B} \Rightarrow x_1, \mathrm{C} \Rightarrow x_2, \mathrm{RESULT} \Rightarrow w' \rangle \rangle$$

Note that we do not make any assumptions about the relationship between $x_0 + x_1$ and w'. This is because just about any relationship we might state (e.g. $|w' - (x_0 + x_1)| < \varepsilon$ for some small ε) will be false on a sufficiently inaccurate machine.

- The corresponding assumption for statement 20, i.e. if no exception occurs, control goes to statement 30, the values of A, B and C don't change and RESULT will be assigned some value. The formal statement is the same as the above, with "10" replaced by "20" and "20" by "30".

- If control is at statement 30 or an exception has occurred, then nothing further happens. Formally, this means that if a trace σ ends in ! or in a state s of the form

  $$\langle 30, [\text{some variable assignment}] \rangle$$

  then σ is maximal, i.e. there is no trace that extends σ and is strictly longer. We also want to assume the converse, i.e. that if σ is maximal then it either ends in ! or in a state with control at statement 30.

The above conditions do not *ensure* that an event system corresponds to an implementation of the program on a finite machine. They are merely a weakening of the conditions we wrote down for the ideal machine which

allow event systems corresponding to finite implementations. In fact, the above conditions are met by the ideal implementation. Since we want to verify the program assuming that it *is* running on a finite machine, we need an additional condition which not only ***allows*** finite implementations, but actually ***rules out*** infinite implementations. The additional condition we will impose is that the set of all events appearing in ***any*** sequence in the event system is finite. This will rule out the infinite implementations.

We will refer to the above conditions on event systems as the ***absolute axioms*** of the program. We call them "absolute" because they are assumed to hold for ***all*** implementations of the program.

We have not yet said what it means for the program to asymptotically compute the 3-ary addition function. Before we do, let's stop and think about what we could possibly verify about how the program runs on an arbitrary finite machine. On the basis of the absolute axioms, we can verify that the program does not go into an infinite loop, i.e. there is no infinite sequence of events such that every finite initial segment is in the event system. (From now on we will refer to such infinite sequences of events as *infinite paths* through the event system). We can verify that if the program does not terminate with an exception, it terminates with RESULT assigned a value, and with A, B and C having the same values they did initially. We cannot verify too much more than that about the program from the assumptions we've made. In fact, it is easy to prove that for any values of x_0, x_1, x_2 and w, there is some event system T satisfying the absolute axioms such that there is some trace in T which starts with inputs x_0, x_1 and x_2 and terminates with output w. Thus we can't prove anything about how well the program computes the 3-ary addition function. The reason for this is that we don't have any conditions on how machine addition is related to ideal addition.

What we want to be able to verify is that if we require that machine addition match ideal addition more and more closely, that we will be able to prove that the input/output behavior of the program matchs the 3-ary addition function more and more closely. In other words, we want to be able to prove that for any desired degree d of accuracy of the 3-ary addition function, there exists a degree of accuracy d' of 2-ary addition such that for any event

system T satisfying the absolute axioms and d', the input/output behavior of T will satisfy d.

What do we mean by a "degree of accuracy"? The intuitive idea is that a degree of accuracy is some condition on implementations of the program which will be met by all sufficiently accurate implementations. Some degrees of accuracy will correspond to the accuracy of the 2–ary addition used by the program; we will refer to these as the *asymptotic axioms* of the program, because they are *assumed* true of all sufficiently large implementations. Other degrees of accuracy will correspond to the accuracy of the 3–ary addition the program is attempting to compute; we will refer to these as the *asymptotic specifications* of the program, because we want to *prove* them about all sufficiently accurate implementations. Formally, a degree of accuracy will be a set of event systems.

What kind of degrees of accuracy do we want to achieve in computing 3–ary addition? What we'd *like* is for every event system T meeting the above conditions to satisfy the following conditions:

1. We can give any input to T, i.e. $\forall x_0, x_1, x_2 \in \mathbf{R}, \exists$ a state of the program s such that $\langle s \rangle \in T$ and s assigns A to x_0, B to x_1 and C to x_2.

2. If $\langle s \rangle \in T$ assigns A to x_0, B to x_1 and C to x_2, then any run of T must eventually terminate normally with RESULT assigned to value $x_0 + x_1 + x_2$. Put more formally, there is no infinite path through T whose first element is s, and for every maximal $\sigma \in T$ whose first element is s, the last element of σ is a state in which control is at statement 30 and RESULT is assigned the value $x_0 + x_1 + x_2$.

Of course, the above conditions can't possibly be satisfied by any such T, if only for the reason that we can't start up a finite implementation of the program with an arbitrary input. We do expect, however, that if we take larger and larger machines, we will be able to approximate fixed inputs with more and more accuracy.

Definition 2.1.1: for any $x_0, x_1, x_2, \delta > 0 \in \mathbf{R}$, we define the degree of

accuracy **inputs**$(x_0 x_1, x_2, \delta)$ to be the set of all event systems T such that $\exists$ a state s such that $\langle s \rangle \epsilon T$ and s assigns A, B and C to numbers y_0, y_1 and y_2 respectively and $|y_i - x_i| < \delta$ for $i = 0, 1, 2$.

□

In general, we won't even be able to get accurate sums of numbers we *can* input to an implementation, due to roundoff, underflow and overflow. To figure out what we can reasonably specify about the program, we must first consider our picture of how such a program is used.

We imagine a "caller" has some inputs x_0, x_1 and x_2 it wishes to submit to the program. Ideally, the caller would like to be able to hand the program x_0, x_1 and x_2 and have it hand it back $x_0 + x_1 + x_2$. In general, the caller will not be able to hand the program x_0, x_1 and x_2, but will have to hand it some *approximations* to these numbers, say y_0, y_1 and y_2, such that $\exists s$ such that $\langle s \rangle \epsilon T$ and s assigns A, B and C the values y_0, y_1 and y_2. The program then "assumes" that y_0, y_1 and y_2 are the inputs the caller is actually interested in. It is the "responsibility" of the program to try and halt with an output which is an "approximation" to $y_0 + y_1 + y_2$. It is the responsibility of the caller to supply the program with sufficiently "good" approximations to justify the program's "assumption". Note that the caller's responsibility is only to give the program inputs which are sufficiently close to x_0, x_1 and x_2; it is not required to give the program *particular* inputs which are sufficiently close.

Fix $x_0, x_1, x_2 \epsilon \mathbf{R}$ and $\varepsilon > 0$. Suppose the caller would be satisfied if the program returned it some number w such that $|w - (x_0 + x_1 + x_2)| < \varepsilon$. How close approximations to x_0, x_1 and x_2 does the caller have to supply in order the get an output in $(x_0 + x_1 + x_2 - \varepsilon, x_0 + x_1 + x_2 + \varepsilon)$? First of all, it must at least supply approximations y_0, y_1 and y_2 such that $|(y_0 + y_1 + y_2) - (x_0 + x_1 + x_2)| < \varepsilon$, because if it did not, the program would be "justified" in handing it back a number close to $y_0 + y_1 + y_2$, possibly so close that it would be more than ε from $x_0 + x_1 + x_2$. Suppose δ is sufficiently small that for any y_0, y_1, y_2 such that $|y_i - x_i| < \delta$ for $i = 0, 1, 2$, $|(y_0 + y_1 + y_2) - (x_0 + x_1 + x_2)| < \varepsilon$ (any $\delta \leq \varepsilon/3$ will do). If the caller limited itself to inputing approximations in which $|y_i - x_i| < \delta$ for $i = 0, 1, 2$, would some sufficiently accurate machine ensure that the answer returned to the

caller is in $(x_0 + x_1 + x_2 - \varepsilon, x_0 + x_1 + x_2 + \varepsilon)$? We cannot really give a firm "yes" or "no" to this question because we do not yet have a formal definition of what "sufficiently accurate" means. The "intuitive" answer, however, seems to be "no". To see why, consider the following example. Suppose we had $x_0 = x_1 = x_2 = 1$, $\varepsilon = 1.5$, and $\delta = .5$. Suppose we were running our program on a machine in which addition was allowed to introduce an absolute error of up to some small number $\xi > 0$, and $.5 + \xi/2$ was representable in the machine. Suppose the caller approximated x_0, x_1 and x_2 by $.5 + \xi/2$. Since the machine is allowed to introduce up to ξ much error when performing an addition, it could assign RESULT to 1 in statement 10. It could then assign RESULT to a number as small as $1.5 - \xi/2$ in statement 20, which is not in $(x_0 + x_1 + x_2 - \varepsilon, x_0 + x_1 + x_2 + \varepsilon)$. We can make the machine we're running the program on arbitrarily accurate by making ξ very small, but by the above argument, there will always be some approximations in the $(x_i - \delta, x_i + \delta)$ intervals which will cause the program to return a value more that ε from the correct answer.

The reason this can happen is that the caller can choose its y_0, y_1 and y_2 just slightly less that δ from the corresponding x_i's. When it does this, $|(y_0 + y_1 + y_2) - (x_0 + x_1 + x_2)|$ is just slightly less than ε. Thus, even a small error in the two machine additions can make RESULT more than ε from the exact answer.

Suppose it were actually the case that for any y_0, y_1 and y_2 such that $|x_i - y_i|$ is less than *or equal to* δ, $|(y_0 + y_1 + y_2) - (x_0 + x_1 + x_2)| < \varepsilon$. Again we pose the informal question: if the caller limited itself to inputing approximations y_i in $(x_i - \delta, x_i + \delta)$, would some sufficiently accurate machine ensure that the answer returned to the caller is in $(x_0 + x_1 + x_2 - \varepsilon, x_0 + x_1 + x_2 + \varepsilon)$? The "intuitive" answer now seems to be "yes". Supporting evidence for this answer is the fact that the answer returned to the caller *will* be within ε of the number we want if we run our program on a machine which uses floating point arithmetic with a sufficiently large number of bits in the mantissa and exponent. (We prove this below.) We can therefore define a degree of accuracy corresponding to all event systems large enough to meet the above condition.

Definition 2.1.2: for every $x_0, x_1, x_2 \in \mathbf{R}$ and $\varepsilon, \delta > 0$ we define the degree

of accuracy accuracy$(x_0, x_1, x_2, \varepsilon, \delta)$ to be the set of all event systems T such that if

$$\forall y_0, y_1, y_2 \in \mathbf{R}[|y_i - x_i| \leq \delta \text{ for } i = 0, 1, 2 \rightarrow$$
$$|(y_0 + y_1 + y_2) - (x_0 + x_1 + x_2)| < \varepsilon]$$

then $\forall s$ such that $\langle s \rangle \in T$ and s assigns A, B and C to numbers y_0, y_1 and y_2 respectively and $|y_i - x_i| < \delta$ for $i = 0, 1, 2$, T must terminate and return a value in $(x_0 + x_1 + x_2 - \varepsilon, x_0 + x_1 + x_2 + \varepsilon)$ (i.e. there are no infinite paths through T which start with s, and if σ is a maximal element of T which starts with s then the last element of σ must be a state which assigns RESULT a value w such that $|w - (x_0 + x_1 + x_2)| < \varepsilon$).

□

The inputs and accuracy degrees of accuracy constitute the asymptotic specifications of our program. The inputs degrees will also be asymptotic axioms. This may seem peculiar, but it just reflects the fact that the ability to approximate fixed inputs more and more closely on bigger and bigger machines is both necessary to asymptotically compute 3-ary addition, and something we can assume is true.

What kind of asymptotic axioms can we assume about the machine's 2-ary addition? We want to assume conditions like the accuracy requirements above, only on 2-ary addition in the middle of the program's execution.

Definition 2.1.3: for any $x_0, x_1 \in \mathbf{R}$ and $\varepsilon, \delta > 0$, we define the degree of accuracy primacc$(x_0, x_1, \varepsilon, \delta)$ to be the set of all event systems T such that if $\forall y_0, y_1$ such that $|y_i - x_i| \leq \delta$ for $i = 0, 1$, $|(y_0 + y_1) - (x_0 + x_1)| < \varepsilon$,

1. if $\sigma \hat{} \langle e, e' \rangle \in T$ and e is a state in which control is at statement 10 and A is assigned a value in $(x_0 - \delta, x_0 + \delta)$ and B is assigned a value in $(x_1 - \delta, x_1 + \delta)$, then e' is a state in which RESULT is assigned a number in $(x_0 + x_1 - \varepsilon, x_0 + x_1 + \varepsilon)$.

2. if $\sigma \hat{} \langle e, e' \rangle \in T$ and e is a state in which control is at statement 20 and RESULT is assigned a value in $(x_0 - \delta, x_0 + \delta)$ and C is assigned a

value in $(x_1-\delta, x_1+\delta)$, then c' is a state in which RESULT is assigned a number in $(x_0+x_1-\varepsilon, x_0+x_1+\varepsilon)$.

□

There is a condition that must hold of the asymptotic axioms in order for them to make sense, namely, for any finite set B of asymptotic axioms there must exist an event system satisfying the absolute axioms which satisfies every $\beta \in B$. If this is not true, then our asymptotic axioms are too strong. When a set of conditions has the property that any finite collection can be satisfied, we will say that the set of conditions is *finitely satisfiable*.

Proposition 2.1.1: The collection of absolute and asymptotic axioms for the program is finitely satisfiable.

Proof: We will show that any degree of accuracy is met by a finite machine which uses binary floating point arithmetic with n bits in the mantissa and n bits in the exponent if n is sufficiently large. Since such machines meet all the absolute axioms, this will establish the proposition.

Suppose the degree in question is inputs(x_0, x_1, x_2, δ). 2^n will be machine-representable. If we take n large enough that 2^n is bigger than the absolute values of all the $x_i \pm \delta$'s, and big enough that the minimum spacing between numbers whose absolute values are $< 2^n$ is $< \delta$, then there will necessarily be a machine-representable real in every $(x_i-\delta, x_i+\delta)$ interval because there will be machine-representable numbers both above and below the interval, and the spacing between machine representable numbers is too small for the interval to be between 2 adjacent machine representable numbers.

Suppose the degree in question is primacc$(x_0, x_1, \varepsilon, \delta)$. The degree is satisfied vacuously unless $2\delta < \varepsilon$; suppose this is the case. If we let n be sufficiently large that the minimum spacing between machine representable numbers in the interval $I = (x_0 + x_1 - \varepsilon, x_0 + x_1 + \varepsilon)$ is less than $\varepsilon - 2\delta$, and $|y_i - x_i| < \delta$, then $y_0 + y_1$ will be in the interval I, so the machine computation of the sum will be off by less than $\varepsilon - 2\delta$. Thus, the difference between the machine sum and the actual sum can be at most the sum of the differences between the y_i's and the x_i's (δ each) and the maximum machine error, $\varepsilon - 2\delta$. In other words, the maximum error is ε, as desired.

Now we are ready to state formally what we wish to mean by asymptotic correctness of the program. We say the program is asymptotically correct iff for every finite set A of asymptotic specifications, there exists a finite set B of asymptotic axioms such that if T is an event system which satisfies the absolute axioms and is in every $\beta \in B$, then T is in every $\alpha \in A$.

Proposition 2.1.2: The program is asymptotically correct.

Proof: We need only show that for any degree accuracy$(x_0, x_1, x_2, \varepsilon, \delta)$, there is a finite set of primacc degrees such that if we assume the program satisfies the finite set of primacc degrees then we can prove it satisfies the accuracy degree. There is a finite set of primacc degrees which ensure that the errors in statements 10 and 20 is less than $(\varepsilon - 3\delta)/2$. The difference between the actual sum $x_0 + x_1 + x_2$ and the final value of RESULT is at most the sum of the differences between the y_i's and the x_i's (δ each) and the sum of the the two computation errors ($(\varepsilon - 3\delta)/2$ each). This adds up to at most ε, as desired.

2.2 Generalized Asymptotic Computation

In this section we generalize the example given in the last section to a general model of asymptotic computation.

2.2.1 Programs

We will first generalize the notion of a program. We define a language of flow chart programs called SRNL for Simple Real Number Language. Before we describe SRNL, we will make the following comment: it's been our experience that in order to write asymptotically correct programs to do nontrivial tasks, it is necessary that we be able to detect exceptional

conditions such as overflow and specify what the program does when such exceptional conditions occur. We have accommodated this necessity in SRNL by incorporating exception-handling. This is discussed further below.

A program consists of:

1. a finite collection of variables
2. an assignment of types to the variables
3. a *flow chart*

We allow variables of types integer (including both positive and negative integers) and real.

A flow chart, as we define it, is a certain kind of directed graph, with the nodes corresponding to points of control within the program and the arrows corresponding to possible flows of control. We will first describe what we mean by a flow chart informally (although the only thing that will be informal about the definition is that it will be in English rather than first-order logic), and then give a formal definition. A flow chart is a finite directed graph in which some of the nodes and arrows may be labeled. Each node is assigned to exactly one of the following categories:

- start nodes (these are the nodes where control can be when the program starts executing)
- halt nodes (these nodes correspond to normal program termination)
- assignment nodes (these are the nodes where variables are assigned new values)
- test nodes (these are the nodes where control branches according to some condition)

Some categories may have no nodes in them, but there must be at least one start node.

Arrows can be unlabeled, or they can be labeled with one of the following labels: "true", "false" or "exception". (Unlabeled arrows correspond to unconditional control flows; arrows labeled with "true" or "false" correspond to conditional control flows; arrows labeled with "exception" correspond to control flows associated with exception handling.)

Start nodes are unlabeled. They may have only unlabeled arrows coming from them, and each start node must have at least one arrow coming from it. Start nodes can have no arrows going to them.

Halt nodes are unlabeled. They may not have any arrows coming from them.

Each assignment node is labeled with an *assignment statement*. An assignment statement is a statement of the form

$$v := t$$

where v is a variable of the program and t is a term whose output type is the same as the type of v. A term is just a program variable, a constant symbol or a function symbol applied to a collection of program variables. We will list the constant and function symbols and their types below. An assignment node must have at least one unlabeled arrow coming from it, and every arrow coming from an assignment node must either be an unlabeled arrow or must be labeled with "exception".

Each test node is labeled with a *boolean expression*. We will define the boolean expressions below. A test node must have at least one arrow labeled "true" and one arrow labeled "false" coming from it, and every arrow coming from a test node must be labeled.

Terms are built up from program variables and constant symbols by applying function symbols. The constant and function symbols (listed by type signature) that we will be using are:

1. constant symbols of type integer: $0_{\mathbf{Z}}$ and $1_{\mathbf{Z}}$
2. constant symbols of type real: $0_{\mathbf{R}}$ and $1_{\mathbf{R}}$
3. binary function symbols which take integers and return integers: $+_{\mathbf{Z}}$, $-_{\mathbf{Z}}$, $*_{\mathbf{Z}}$
4. binary function symbols which take integers and return integer: $+_{\mathbf{R}}$, $-_{\mathbf{R}}$, $*_{\mathbf{R}}$, /

Note: in actual examples, we will "cheat" in a couple of ways to make our programs more readable. For example, technically, we need subscripts on symbols like "1" and "+" to distinguish between integer constants and functions and real constants and functions which are usually denoted by the same symbol. In our examples, we will drop the subscripts, and it will always be clear from context whether we mean the integer symbols or the real ones. Also, we will use other symbols besides those above, e.g. other numerals, like "2", and the unary $-$ function. These symbols should be regarded as abbreviations for terms written using only the symbols above, so "2" is an abbreviation for "1+1" and "$-x$" is an abbreviation for "$0-x$". Finally, we will use more complex terms in our assignment statements than just the simple terms allowed by SRNL. These terms are abbreviations for pieces of code which evaluate the complex expression one subterm at a time, storing the intermediate results in temporary variables. The restriction to simple terms will eventually be removed from SRNL, but for the time being we have placed this limitation on the programs to make the semantics easier to state. The principal difficulty in stating semantics for complex terms is that an exception may occur in the middle of evaluating a term, which can't happen with the simple terms we're restricting ourselves to at the moment.

Boolean expressions are built up from atomic boolean expressions by applying boolean connections. We allow all the usual boolean connectives (e.g. $\wedge, \vee, \neg, \rightarrow$). Atomic boolean expressions are of the form

$$P(v_0, \ldots, v_{n-1})$$

where P is a predicate symbol and $v_0, \ldots, v_{n-1}$ are program variables whose types match the input type signature of P. The predicate symbols we will use, listed by type signature, are:

1. binary predicate symbols which take integer arguments: $=_{\mathbf{Z}}, <_{\mathbf{Z}}$
2. binary predicate symbols which take real arguments: $=_{\mathbf{R}}, <_{\mathbf{R}}$

(Again, in actual examples we will drop the subscripts, and will use abbreviations like "$x \leq y$" for "$x = y \vee (x < y)$".)

We will now give the formal definition of a flow chart. A flow chart is an 12-tuple $(N, \text{START}, \text{HALT}, \text{ASSIGN}, \text{TEST}, \text{UA}, \text{TA}, \text{FA}, \text{EA}, \text{AL}, \text{TL}, \text{L})$ such that:

1. N is a nonempty set (the *nodes*).
2. START, HALT, ASSIGN and TEST are disjoint subsets of N whose union is all of N. START is nonempty.
3. UA, TA, FA and EA are binary relations on N (UA is the *unlabeled arrow relation*, TA is the *"true" arrow relation*, FA is the *"false" arrow relation* and EA is the *"exception" arrow relation*).
4. L is a function from ASSIGN $\cup$ TEST into the set consisting of the assignment statements and boolean expressions of the program (the *label function*). $\forall \alpha \in \text{ASSIGN}$, $\text{L}(\alpha)$ is an assignment statement. $\forall \alpha \in \text{TEST}$, $\text{L}(\alpha)$ is a boolean expression.
5. $\forall \alpha, \beta \in N$, if $\text{UA}(\alpha, \beta)$ or $\text{TA}(\alpha, \beta)$ or $\text{FA}(\alpha, \beta)$ or $\text{EA}(\alpha, \beta)$ then $\alpha \notin \text{HALT}$ and $\beta \notin \text{START}$.
6. $\forall \alpha, \beta \in N$, if $\text{UA}(\alpha, \beta)$ then $\alpha \notin \text{TEST}$.
7. $\forall \alpha, \beta \in N$, if $\text{TA}(\alpha, \beta)$ or $\text{FA}(\alpha, \beta)$ then $\alpha \in \text{TEST}$.
8. $\forall \alpha, \beta \in N$, if $\text{EA}(\alpha, \beta)$ then $\alpha \notin \text{START}$.
9. $\forall \alpha \in \text{START} \cup \text{ASSIGN}$ $\exists \beta \in N$ such that $\text{UA}(\alpha, \beta)$.
10. $\forall \alpha \in \text{TEST}$ $\exists \beta, \gamma \in N$ such that $\text{TA}(\alpha, \beta)$ and $\text{FA}(\alpha, \gamma)$.

2.2.2 Semantics

We want to give a semantics to the programs defined in the previous subsection by associating each program with a class of event systems. The various event systems in the class correspond to implementations of the program on machines of various sizes. We will now fix a program P and describe the class of event systems associated with it. The members of this class will be referred to as the *models* of the program. We will denote the set of nodes, categories of nodes, arrow relations and label function using the same notation as in the previous subsection.

First, we need to say what a state of P is. A state consists of:

1. a node of the flow chart (this represents the place where control is resting)
2. an assignment of some subset of the variables to elements of their *associated type*

Formally, then, a state is a pair $\langle \alpha, V \rangle$ where α is a node and V is a function from some subset of the program variables into the disjoint union of $\mathbf{Z}$ and $\mathbf{R}$ which takes integer variables to integers and real variables to real numbers.

If s is a state and v a variable which is assigned a value by s, we will denote the value assigned to v by s by $s(v)$. If t is a term all of whose variables are assigned a value by s, we denote the ideal value of the term under this assignment by $s(t)$.

A model is an event system over the set of states described above which meets certain conditions. As in the previous section, these conditions will be referred to as the absolute axioms of P. We will first discuss certain considerations which influenced the conditions we impose, then we will state the conditions informally, and then give their formal equivalents.

In formulating the conditions for an event system being a model of P, the following considerations were taken into account:

1. Nothing was assumed about the accuracy of real-valued functions. This was because the conditions we give below are intended to define what we felt we could assume about P running on *any* machine, whether large or small. Almost any assumption about accuracy of real-valued functions would be invalid on a sufficiently small machine.

2. Integer-valued functions, by contrast, were assumed to be perfectly accurate when they did not cause an exception to be raised. We felt this was a reasonable assumption on both large and small machines.

3. We did not assume that there were any circumstances in which the evaluation of a real- or integer-valued function would not raise an exception. In other words, we allow "maximum flakiness" from the real- and integer-valued functions. This was, again, because just about any assumption about functions not raising exceptions would be invalid on a sufficiently small machine.

4. Comparisons of numbers (i.e. for equality or $<$) were assumed to be accurate, and furthermore were assumed *not to raise exceptions*.

5. Assignments of the form $v := w$ where w is a program variable were assumed to be perfectly accurate. In other words, it was assumed that error only arises from evaluating arithmetic functions, and not from copying values of variables into other variables.

6. Other things assumed to be carried out accurately were evaluation of boolean connectives, detection of undefined variables, flow of control, and holding constant the values of variables not assigned to.

The informal statements of the conditions are as follows:

1. Initially, control is always at some start node in the flow chart.

2. If control is at a start node α, control flows along some arrow from α, and the values of the variables do not change.

3. Exceptions can only occur at assignment nodes and test nodes.

4. An exception will occur at an assignment or test node if the node's label contains a variable which is not defined.

5. An exception will not occur at an assignment node if the nodes label is $v := w$ and w is a program variable which is defined.

6. An exception will not occur at a test node if all variables in the node's boolean expression are defined.

7. If control is at a node α and an exception occurs, and there are no exception arrows coming from α, then P terminates abnormally.

8. If control is at a node α, and an exception occurs, and there are exception arrows coming from α, then control flows along one of the exception arrows, and the assignment of variables is unchanged.

9. If control is at an assignment node α labeled with assignment statement $v := t$ and no exception occurs, control flows along one of the unlabeled arrows from α, v is assigned a value, and the values of variables other than v do not change. In addition, if t is an integer term and no exception occurs, v is assigned the value $s(t)$. If t is a program variable then v is assigned the value $s(t)$.

10. If control is at a test node and no exception occurs, control flows along an arrow labeled "true" if α's boolean expression is true, and along an arrow labeled "false" if α's boolean expression is false.

11. If control is at a halt node, no further state transitions can occur. If control is *not* at a halt node and P has *not terminated abnormally*, then further state transitions *must* occur.

We now state the formalization of the conditions for an event system T to be a model of P:

1. $\forall \sigma \in T$, ! does not occur twice consecutively in σ.

2. $\forall e$, if $\langle e \rangle \in T$ then $e = \langle \alpha, V \rangle$ for some $\alpha \in$ START.

3. $\forall \sigma, \alpha \in N$ and V an assignment of program variables, if $\sigma \hat{} \langle \langle \alpha, V \rangle, ! \rangle \in T$ then all of the following are true:

   (a) $\alpha \in$ ASSIGN $\cup$ TEST

   (b) It is not the case that $\alpha \in$ ASSIGN, $L(\alpha) =$ "$v := w$" where w is a program variable, ad V assigns a value to w.

   (c) It is not the case that $\alpha \in$ TEST, and V assigns a value to every variable in $L(\alpha)$.

4. $\forall \sigma, \alpha, \beta \in N$ and V, V' assignments of program variables, if

$$\sigma \hat{} \langle \langle \alpha, V \rangle, !, \langle \beta, V' \rangle \rangle \in T$$

   then $EA(\alpha, \beta)$ and $V' = V$.

5. $\forall \sigma, \alpha, \beta \in N$ and V, V' assignments of program variables, if

$$\sigma \hat{} \langle \langle \alpha, V \rangle, \langle \beta, V' \rangle \rangle \in T$$

   then all of the following are true:

   (a) If $\alpha \in$ START then $UA(\alpha, \beta)$ and $V' = V$.

   (b) If $\alpha \in$ ASSIGN and $L(\alpha) =$ "$v := t$" then:

      i. V assigns a value to all variables occurring in t.

      ii. $UA(\alpha, \beta)$, $V'(v)\downarrow$, and $\forall$ program variables $v' \neq v$, $V'(v') \simeq V(v')$.

      iii. If t is an integer term, $V'(v) = V(t)$.

      iv. If t is a program variable, $V'(v) = V(t)$.

   (c) If $\alpha \in$ TEST then:

      i. V assigns a value to all variables occurring in $L(\alpha)$.

      ii. $V' = V$

      iii. $TA(\alpha, \beta)$ if $L(\alpha)$ is true and $FA(\alpha, \beta)$ if $L(\alpha)$ is false.

6. $\forall \sigma \in T$, if the last element of σ is $\langle \alpha, V \rangle$ then σ is maximal in T iff $\alpha \in$ HALT.

7. $\forall \sigma \in T$, if σ is maximal and the last element of σ is !, then $\exists \alpha \in N$ and V an assignment of program variables such that $\langle \alpha, V \rangle$ is the next-to-last entry of σ and $\exists \beta \in N$ such that $\mathrm{EA}(\alpha, \beta)$.

8. The set of all events which appear in some $\sigma \in T$ is finite.

2.2.3 Asymptotic Specifications

In this subsection we generalize the notion of asymptotic specifications from the previous section. As in the previous section, the asymptotic specifications will be a set of degrees of accuracy, but we're going to want to be able to specify something slightly more general about our program P than simply that it asymptotically compute a function. In general, we're going to want to specify that a certain relation hold between the assignment of the variables when P starts and when it ends. We call such a relation a *specification relation* for P.

If R is the binary relation we would like to have hold between the variable assignments at start and termination, we'd *like* to require the following:

1. We can start up P with any assignment of variables.

2. If we start up P with an assignment of variables V, and there exists an assignment of variables W such that $R(V, W)$, then P eventually terminates with an assignment of variables W' (possibly $\neq W$) such that $R(V, W')$.

3. If we start up P with an assignment of variables V and there is no assignment of variables W such that $R(V, W)$, then P either doesn't terminate, or terminates abnormally (i.e. with an exception).

Of course, as in the previous section, we can't in general meet the above requirements on a finite machine. What we will try to verify instead is that for any degree of accuracy d of satisfying R (in the sense described above), there exists a degree of accuracy d' of computing the primitive functions

of SRNL such that for any event system T satisfying the absolute axioms and d', the input/output behavior of T will satisfy d. The remainder of this subsection is devoted to deciding what we want to mean by "degrees of accuracy of satisfying R", and what kinds of R's we will allow ourselves to use in specifications. As before, a degree of accuracy is formally a set of event systems.

Suppose that the real variables of P are $X_1, \ldots, X_n$, and the integer variables are $I_1, \ldots, I_m$. First of all, we can't start up P with an arbitrary assignment of variables on a finite machine. If, however, we have a fixed assignment of variables V, then on a sufficiently accurate machine we want to be able to start up P with a variable assignment V' which is "close to" V. In order to make precise what we mean by "close", we need some notion of "the distance between two variable assignments". If V and V' are two variable assignments, we define the distance between them (denoted by $\rho(V, V')$) as follows:

- If V and V' make the same variables defined and undefined, then $\rho(V, V')$ is the largest element of the set

$$\{|V(v) - V'(v)| \mid v \text{ is a variable defined by both } V \text{ and } V'\}$$

- $\rho(V, V') = 1$ otherwise.

The first clause says that if two variable assignments assign the same set of variables then their distance apart depends on how far apart their assignments of the variables are. The second clause of the definition says essentially that variable assignments which do not assign values to the same set of variables are not "close to" each other.

If V is a fixed variable assignment and $\delta > 0$, then on a sufficiently accurate machine we want to be able to start up P with an assignment of variables V' such that $\rho(V, V') < \delta$. For each assignment V and $\delta > 0$ we can therefore define a degree of accuracy consisting of those event systems which are accurate enough to meet the above condition.

Definition 2.2.1: for any assignment of program variables V and $\delta > 0$ we define the degree of accuracy **startup**(V, δ) to be the set of all event systems T for P such that $\exists$ an assignment of program variables V' such that $\rho(V', V) < \delta$.

□

The **startup** degrees are analogous to the **inputs** degrees of the previous section.

Again, we imagine P being used by a caller which wants to run the program with an initial assignment of variables V and have it terminate with an assignment of variables W such that $R(V, W)$. What can we reasonably specify about how accurately P meets R? Our answer to this question will be complicated somewhat by the fact that there may be *no* W such that $R(V, W)$. We will put off dealing with this complication until later, and for the moment we will assume that there exists W such that $R(V, W)$. We will refer to such W's as *good* variable assignments ("good" in the sense that they are the variable assignments the caller would like to get close to). We will refer to the set of V's such that $\exists W$ such that $R(V, W)$ as the *domain* of R, denoted by dom(R).

In the example, the caller had to run P on a sufficiently large machine and give the program an input sufficiently close to the desired input that it would get an output less than a certain error from the value of the 3 ary addition function. In the more general case we're dealing with here, there may be a number of different good W's, and the caller just wants P to terminate with some assignment of variables which is close to *one of* the good W's. Suppose the caller would be satisfied if the program terminates with variable assignment W' that is within ε of some good W. How good an approximation V' to V does the caller need to start up P with in order to get such a W'? We claim the caller must at least start up P with a variable assignment V' such that:

1. $V' \in$ dom(R)
2. $\forall U$ such that $R(V', U)$, there exists a good W within ε of U

Why must these conditions be met? First of all, suppose the caller started up P with a $V' \notin \mathrm{dom}(R)$. P "assumes" that V' is the variable assignment the caller is actually interested in. Since there is no way it can terminate with a U such that $R(V', U)$, the program would be "justified" in terminating with an exception or not terminating at all. Suppose the caller started up P with a $V' \epsilon \mathrm{dom}(R)$ but there is some U such that $R(V', U)$ and there is no good W within ε of U. The program would be justified in terminating with a variable assignment very close to U, possibly so close that it is not within ε of any good W. Thus, the caller must pick some $\delta > 0$ such that $\forall V'$ such that $\rho(V, V') \leq \delta$, the above two conditions are met, and restrict itself to starting up with variable assignments $< \delta$ away from V. (We require that the two conditions hold for all V' less than *or equal to* δ away from V because otherwise there may exist V' which is just slightly less than δ away from V such that slight errors in the program's arithmetic are just enough to allow the program to terminate with a variable assignment slightly more that ε away from the nearest good W. This situation was illustrated concretely in the Motivating Example).

What if there is no such δ? Consider the following example: suppose our specification is that if we start up the program with $\mathrm{X}_1 = x$ we want it to terminate with $\mathrm{I}_1 = 0$ if $x < \sqrt{2}$ and with $\mathrm{I}_1 = 1$ otherwise. In other words, we want the program to tell us if $x < \sqrt{2}$ or not. Suppose the x the caller is interested in is actually $\sqrt{2}$; then the "good" W's are the ones in which $\mathrm{I}_1 = 1$. No matter how small we take δ, however, there will be some y within δ of $\sqrt{2}$ such that $y < \sqrt{2}$. Even if we ran P on a very accurate machine, if we gave it an input $y \epsilon (\sqrt{2} - \delta, \sqrt{2})$, the program would be justified in terminating with $\mathrm{I}_1 = 0$. (In fact, for such a y, this is the right answer).

There is another way the desired δ can fail to exist. Suppose our specification is that if we start up the program with the value of $\mathrm{X}_1 = x$ and x is a real number which has only 0's after the decimal point, then we want the program to terminate with $\mathrm{I}_1 =$ the integer corresponding to x; otherwise, we want P to either raise an exception or fail to terminate. In terms of binary relations on variable assignments, we want the starting and ending assignments of variables to satisfy R where $R(V, W)$ iff $V(\mathrm{X}_1)$ has only 0's after the decimal place and $W(\mathrm{I}_1)$ is the integer corresponding to $V(\mathrm{X}_1)$.

Suppose the r the caller is interested in is 1; then the specification says P should terminate with $I_1 = 1$. No matter how small we take δ, there will be some y within δ of 1 such that y does not correspond to an integer, and so the program would be justified is terminating with an exception or not terminating.

We didn't encounter this problem in the Motivating Example because 3 ary addition is a continuous, total function, so the δ we need always exists. In the first example above, we are asking P to compute a discontinuous function. In the second example, we are asking P to compute a function for which there are points x in the domain such that there are points y *not* in the domain arbitrarily close to x. In topology, a set O which has the property that if $x \in O$ then $\exists \delta > 0$ such that every y within δ of x is in O is called an *open* set; in the second example, we are asking P to compute a function whose domain is not open.

What all this adds up to is that we can only expect to asymptotically compute functions which have open domains and which are continuous on their domains. We must therefore restrict ourselves to specifying that P asymptotically compute a function F only if F is continuous on an open domain. We can express this in the more general setting of specification relations by restricting ourselves to relations R such that

$$\forall V \in \mathrm{dom}(R), \varepsilon > 0, \exists \delta > 0, \forall V'[\rho(V, V') \leq \delta \rightarrow V' \in \mathrm{dom}(R) \wedge \forall U[R(V', U) \rightarrow \exists W[R(V, W) \wedge \rho(W, U) < \varepsilon]]]]$$

Given that we restrict ourselves to such R's, we define the following degrees of accuracy:

Definition 2.2.2: for any variable assignment V and $\varepsilon, \delta > 0$, we define the degree of accuracy $\mathsf{accuracy}_R(V, \varepsilon, \delta)$ to be the set of all event systems T for P such that if $V \in \mathrm{dom}(R)$ and

$$\forall V'[\rho(V, V') \leq \delta \rightarrow V' \in \mathrm{dom}(R) \wedge \forall U[R(V', U) \rightarrow \exists W[R(V, W) \wedge \rho(U, W) < \varepsilon]]]$$

then $\forall \epsilon$ such that $\langle \epsilon \rangle \in T$ and the variable assignment associated with ϵ is V' and $\rho(V, V') < \delta$, if T is started up with ϵ then it must eventually terminate normally in a state ϵ' with associated variable assignment U such that $\exists W[R(V, W) \wedge \rho(U, W) < \varepsilon]$ (i.e. there is no infinite path through T which starts with ϵ, and any maximal $\sigma \in T$ which starts with ϵ ends with an ϵ' meeting the above condition).

□

We now return to the question of what we can reasonably specify about how accurately P meets R in the case where the caller is interested in starting up P with an input $V \notin \mathrm{dom}(R)$. Unfortunately, we don't have a good answer to this question at this point. We'd *like* it to be the case that if we take a sufficiently accurate implementation and start it up with a V' sufficiently close to V, that the program will terminate abnormally (i.e. with an unhandled exception) or at least go into an infinite loop. This specification is unfortunately too strict. Consider the following: suppose our specification is that if we start up P with $X_1 = x \neq 0$ then the program terminates with $X_2 = 1/x$. In other words, P computes the reciprocal function. Suppose the x the caller is interested in is 0. Suppose that this expression is being evaluated on a very accurate machine which uses some sort of floating-point representation of reals such that for any $y \neq 0$ representable in the machine, $1/y$ is between two numbers representable in the machine. The caller could input a number very close to 0 and still not get an exception or go into an infinite loop. In fact, one can imagine arbitrarily accurate machines of this sort and inputs arbitrarily close to 0 which would not raise an exception or fail to terminate. Thus, even on a very accurate machine, the caller cannot choose a number sufficiently close to 0 that will cause the program to indicate that the expression the caller is actually trying to evaluate (i.e. $1/0$) is undefined.

Our "solution" to this problem at the present time is to define asymptotic computation solely in terms of what P does when started up with a variable assignment V' which is "near" a $V \in \mathrm{dom}(R)$. In other words, we use the degrees of accuracy defined above, which only concern accuracy of computation on V's in $\mathrm{dom}(R)$. Thus, with our present definition, proving asymptotic correctness of a program does not tell us anything about what

kind of behavior we can expect if we run the program on larger and larger machines with starting variable assignments closer and closer to $V \notin \mathrm{dom}(R)$. This is not really an acceptable solution: we are still working on the problem.

The asymptotic specifications are therefore the degrees $\mathsf{startup}(V, \delta)$ and $\mathsf{accuracy}(V, \varepsilon, \delta)$.

2.2.4 Asymptotic Axioms

Our asymptotic axioms will be statements of the same form as the $\mathsf{accuracy}_R$ degrees about the execution of the program's primitive functions. We can simplify the definition somewhat since the asymptotic axioms are just specifying that certain functions are computed accurately (rather than some more complicated specification in terms of a binary relation on variable assignments).

Definition 2.2.3: for any assignment node a with label $v := F(v_1, \ldots, v_l)$ and variable assignment V and $\varepsilon, \delta > 0$, we define the degree of accuracy $\mathsf{primacc}_a(V, \varepsilon, \delta)$ to be the set of all event systems T such that if V assigns a value to $v_1, \ldots, v_l$ and $F(V(v_1), \ldots, V(v_n))\downarrow$ and

$$\forall V'[\rho(V, V') \leq \delta \rightarrow F(V'(v_1), \ldots, V'(v_l))\downarrow \wedge$$
$$|F(V'(v_1), \ldots, V'(v_l)) - F(V(v_1), \ldots, V(v_l))| < \varepsilon]$$

then $\forall \sigma, e, e'$, if $\sigma \hat{} \langle e, e' \rangle \in T$ and $e = \langle a, V' \rangle$ and $\rho(V, V') < \delta$ then $e' \neq !$ and e' assigns v a value w such that

$$|w - F(V(v_1), \ldots, F(v_l))| < \varepsilon$$

□

The asymptotic axioms are the degrees $\mathsf{startup}(V, \delta)$ and $\mathsf{primacc}_a(V, \varepsilon, \delta)$ (as in the Motivating Example, the degrees which say we can approximate

inputs closely are both part of the specification and something we can assume).

Note that the $\mathsf{primacc}_\alpha$ degrees don't merely restrict how bad roundoff error, etc. can be; they also restrict the circumstances under which exceptions can occur. The absolute axioms place almost no restrictions on when exceptions can occur. In fact, an event system may raise an exception on *every* assignment statement and still satisfy the absolute axioms. If we require that more and more asymptotic axioms are met, however, we find that the circumstances in which an event system is allowed to raise an exception are more and more restricted.

We need to check that the asymptotic axioms are finitely satisfiable. It is easy to see that for any finite collection A of asymptotic axioms there exists an event system for P which meets the absolute axioms and every axiom in A, just by taking:

1. a sufficiently large initial segment of the integers as the machine's integer type
2. real numbers expressible in binary floating–point notation with a sufficiently large exponent and mantissa (this only one of many choices one could make) as the machine's real number type
3. integer arithmetic is exact unless it takes us outside the integer type (in which case raise an exception)
4. real arithmetic rounds to the nearest number in the real number type unless it takes us above the largest positive machine–representable number or below the largest negative machine–representable number (in which case raise an exception).

The proof of the last statement is completely analogous to the proof of finite satisfiability in the Motivating Example, so we omit it.

Having stated the asymptotic specifications and axioms, we can now make the following definition:

Definition 2.2.4: A program P *asymptotically satisfies* a specification relation R iff for every finite set A of asymptotic specifications for R, there exists a finite set of asymptotic axioms B such that for every model T of P, if T is satisfies every axiom in B then T satisfies every specification in A.

□

Chapter 3

Nonstandard Formulation of the Theory

The Theory of the previous chapter was entirely formulated in the language of classical analysis. In this Chapter we give a formulation of the Theory in Nonstandard Analysis.

3.1 Nonstandard Mathematics

Nonstandard analysis is an alternate approach to doing real analysis. It uses formalizations of intuitive concepts like "infinitesimal" in place of classical methods using limits (the so called $\varepsilon - \delta$ approach).

When calculus was first developed by Newton and Leibniz, the proofs were presented in terms of "infinitesimal" quantities. For instance, the derivative of x^2 was computed by forming the difference quotient

$$\frac{(x + dx)^2 - x^2}{dx}$$

with dx being an infinitesimal quantity. This simplifies by simple algebra

to $2x + dx$. Disregarding the infinitesimal, the derivative, $2x$, is obtained. An infinitesimal was a number which was positive, but less than any *given* positive number. On the face of it, this is inconsistent, since if dx is positive, it must be less than itself. Attempts to make this approach rigorous failed, and methods involving limits were developed. To compute the derivative of x^2 using limits, instead of using a single infinitesimal dx, one examines what happens to the quantity

$$\frac{(x + \Delta x)^2 - x^2}{\Delta x}$$

as smaller and smaller values of Δx are plugged in. One can show that the Δx difference quotient can be made as close as one wants to $2x$ by constraining Δx to be less than a certain size.

In the 1960's logicians developed a way to make the methods of Newton and Leibniz rigorous, and the resulting alternate approach to analysis is called Nonstandard Analysis. (We prefer the term "Nonstandard Mathematics", since the methods used are applicable in other areas of mathematics besides analysis, and will use this term hereinafter). The essential feature of Nonstandard Analysis is the addition of "nonstandard elements" of the domain of discourse which are then used to prove results about the original, standard domain. In this way it is similar to the construction of the complex numbers from the reals. To get the complex numbers, one simply adds a new number denoted by i, assumes that it obeys the axiom $i^2 = -1$ plus the algebraic laws of the reals (e.g. commutativity of addition), and computes with it formally. The resulting extended number system, the complex numbers, can then be used to obtain easier proofs of results about the reals, like the Fundamental Theorem of Algebra.

Let's consider what it would mean to add an infinitesimal to the standard real numbers in a purely formal way. Let L be the first-order language with the following symbols:

- For each n-ary predicate P over $\mathbf{R}$, we have an n-ary predicate symbol $\boldsymbol{P}$ whose interpretation is P.

- For each n-ary function f over $\mathbf{R}$, we have an n-ary predicate symbol $\boldsymbol{f}$ whose interpretation is f.
- for each real number r we have a constant symbol $\boldsymbol{r}$ whose interpretation is r.

Let T be the set of all first-order statements in the language L which are true. T is what is called the *complete theory* of $\mathbf{R}$ over L. Intuitively, it contains all the first-order facts about $\mathbf{R}$.

Next, we add a new symbol ε to our language. This is intended to be the name of the infinitesimal we're adding. We also add to T the axiom $\varepsilon > 0$. We'd like the add an axiom which says that ε is less than any given positive real number. If we add the axiom $\forall x > 0, \varepsilon < x$ we get an inconsistent system, because ε itself is positive. We don't really want this axiom though; what we really want to assume is that $\varepsilon < x$ for all of the *standard* x's, i.e. the real numbers that we started with. We can do this in a formal way by adding a separate axiom $\varepsilon < \boldsymbol{r}$ for every standard positive r. Call the resulting system T'.

Lemma 3.1.1: T' is consistent.

Proof: This follows essentially from the fact that first-order logic is a finitary logic, i.e. every proof is a finite derivation from a finite set of axioms. Therefore, if there's a proof of a contradiction from a set of axioms. there must be a proof of a contradiction from some finite subset of the axioms. Suppose T' were inconsistent. There must then be a certain finite set $T_0 \subseteq T$ and a finite set of positive reals $r_1, \ldots, r_n$ such that there is a proof of contradiction from the axioms of T_0 plus $\varepsilon > 0$ plus the axioms $\varepsilon < \boldsymbol{r_i}$ for $i = 1, \ldots, n$. Since first-order logic is sound, this means that there is no model for this finite set of axioms. Suppose, however. that we interpret the symbol ε to be the real number obtained by taking the smallest of the r_i and dividing it by 2. This interpretation makes all the axioms of the finite subset true. Therefore, no finite subset of T' is inconsistent, so T' as a whole is not inconsistent.

■

By the completeness of first-order logic, any consistent theory has a model. Thus, there exists some model of T'. Call this model M. Since our language has a constant symbol $\boldsymbol{r}$ for every $r \in \mathbf{R}$, and since every such constant symbol has an interpretation $M(\boldsymbol{r})$ in M, we can embed $\mathbf{R}$ into M by the mapping $r \mapsto M(\boldsymbol{r})$; without loss of generality, we can identify $\mathbf{R}$ with its image in M and just assume that $\mathbf{R} \subseteq M$. We know that there is at least one element of M which is not in $\mathbf{R}$, namely the interpretation of ε. Also, since M is a model of T' we know that ε is a positive number which is less than every positive number in $\mathbf{R}$. Thus, we have added an infinitesimal. Also, our new number system M has all the same first order properties that $\mathbf{R}$ does, because M is a model of T, the complete first-order theory of $\mathbf{R}$. Also, every predicate and function on $\mathbf{R}$ has an extension to M. Such extensions of one model by another that preserve all first-order properties are called *elementary extensions*.

The construction of a nonstandard extension of the reals is actually slightly more complicated. The thing that made the above construction work is that we had an infinite collection of "requirements" on ε that were finitely satisfiable in the original, standard reals. In other words, we wanted ε less than every standard positive real number, and for any finite set of positive real numbers there is a *standard* real that is less than everything in the finite set. This finite satisfiability allowed us to show that any finite subset of the theory T' had a model, and was therefore consistent, and so T' was consistent. In fact, we could do the above construction for any collection of requirements which was finitely satisfiable. If the requirements cannot *all* be satisfied in the standard reals, as was the case in the above construction, the M we get is a proper elementary extension of $\mathbf{R}$ in which the collection of requirements *is* satisfiable by a single, nonstandard number.

In the actual construction of a nonstandard extension of $\mathbf{R}$, we do the above construction for *all* finitely satisfiable collections of requirements at once. Before giving the construction, we say precisely what we mean by a finitely satisfiable collection of requirements.

Definition 3.1.1: If $x_1, \ldots, x_n$ are variables in the language of L, a *collection of requirements over* $x_1, \ldots, x_n$ is a set F of formulas in the language L such that for every $\phi \in F$, the free variables of ϕ are among $x_1, \ldots, x_n$. F

is *finitely satisfiable* iff for every finite set

$$\{\varphi_1(x_1,\ldots,x_n),\ldots,\varphi_m(x_1,\ldots,x_n)\}$$

of formulas of F, there exists $r_1,\ldots,r_n \in \mathbf{R}$ such that for $i = 1,\ldots,m$, $\varphi_i(r_1,\ldots,r_n)$ is true.

□

Let L and T be as before. For every finitely satisfiable collection of requirements F on variables $x_1,\ldots,x_n$, we add distinct constant symbols $c_{F,1},\ldots,c_{F,n}$ to the language. We then add to T the axioms $\phi(c_{F,1},\ldots,c_{F,n})$ for every $\phi \in F$. Call the resulting theory T'. Every finite set $T_0 \subseteq T'$ involves at most finitely many F's, and for each such F, there are only finitely many ϕ's from F in the subset. By the finite satisfiability of the F's, we can interpret all the new constants in T_0 in $\mathbf{R}$ so as to make all the axioms of T_0 true. Thus, T' is consistent, and so has a model M. As before, $\mathbf{R}$ can be considered to be a subset of M. M is an elementary extension of $\mathbf{R}$, and every collection of requirements which is finitely satisfiable in $\mathbf{R}$ is actually satisfiable in M (i.e. there exist elements of M making *all* the formulas in the collection true).

We can apply this construction to other sets besides $\mathbf{R}$. In fact, we can apply it to any set.

3.2 Axiomatizing Nonstandard Mathematics

Nonstandard methods can be applied by reasoning about nonstandard models. It is desirable, however, to have an axiomatic system for nonstandard mathematics. Such a system is developed in [2]. We have been using the system in [2] as the basis for our verifications. We now describe it.

Zermelo-Fraenkel set theory with the axiom of choice (usually abbreviated

ZFC) is an axiomatic system in which all standard mathematics can be done. The language of ZFC has only two symbols: a binary predicate symbol for equality and a binary predicate symbol ϵ for set membership (i.e "$x \epsilon y$" means "x is an element of y"). We can therefore think of a model of ZFC as a "universe" for standard mathematics. A nonstandard extension of such a model should then be a "universe" for nonstandard mathematics. In [2], an axiom system for these nonstandard universes called IST is formulated. IST is obtained as follows: starting with a given model M of ZFC, one constructs a nonstandard extension of M'. In M', there is an interpretation of ϵ which satisfies all of the axioms of ZFC. We then add a unary predicate "st" to the language, and interpret it in M' as the set of all elements in M. Finally, we examine what useful axioms in the language of $=$, ϵ and st hold in an arbitrary such M'. IST consists of all the axioms of ZFC plus the additional axioms covering nonstandard mathematics. These additional axioms are presented in 3 schemas. They are:

1.

$$\forall^{\text{st}} x_1, \ldots, x_n [\phi \leftrightarrow \phi^{\text{st}}]$$

where ϕ is an arbitrary formula with no occurrences of the predicate st, $x_1, \ldots, x_n$ includes *all* the free variables of ϕ, and ϕ^{st} means ϕ with every quantifier $\forall y$ replaced by $\forall^{\text{st}} y$ and every quantifier $\exists y$ replaced by $\exists^{\text{st}} y$. What this schema is expressing axiomatically is the fact that M' is an elementary extension of M. It says that if we have any formula in the language of standard mathematics containing standard parameters, then it holds in the nonstandard universe (i.e. ϕ holds) iff it holds in the standard universe (i.e. ϕ^{st} holds). This schema is called the ***transfer principle***. (Formulas which contain no occurrences of the st predicate are called *internal* formulas).

2.

$$\forall^{\text{st finite}} z, \exists x, \forall y \epsilon z, \phi(x, y) \leftrightarrow \exists x, \forall^{\text{st}} y, \phi(x, y)$$

where ϕ is an internal formula in which z does not occur free. What this schema is expressing is the fact that every finitely satisfiable collection of requirements from the standard universe on a single variable

x is satisfied in the nonstandard universe. We think of the formula $\phi(x, y)$ as defining an infinite collection of requirements on x, indexed by standard elements y. The left hand side of the schema says the this collection is finitely satisfiable in the standard universe, i.e. for all finite sets z of standard elements, there is a single x which satisfies the requirements $\{\phi(x, y) \mid y \in z\}$. The right hand side says there is a single x which satisfies *all* of the requirements $\{\phi(x, y) \mid y \text{ is standard}\}$. The schema states that the two are logically equivalent. This schema is called the *principle of idealization.*

3.

$$\forall^{\text{st}} x, \exists^{\text{st}} y, \forall^{\text{st}} z, [z \in y \leftrightarrow z \in x \ \wedge \ \phi(z)]$$

where ϕ is any formula in which y does not occur free (but st can occur in ϕ. This axiom essentially expresses the fact that any collection of standard elements that we can define (even using nonstandard methods) has a standard "extension" in the nonstandard universe. This schema is called the *principle of standardization.*

In [2] an important theorem is proved, namely that IST is conservative over ZFC. What this means is that any statement in the language of ZFC (i.e. no occurrences of "st") which we can prove in IST can be proved from ZFC alone. This tells us that the use of nonstandard methods doesn't change the underlying standard universe. Since the standard world is what we're really interested in, this result is essential.

3.3 Nonstandard Formulation of the Theory

One of the most attractive features of nonstandard mathematics is that definitions become simpler and more intuitive. For example, the classical definition of a sequence of reals $\{x_i \mid i = 0, 1, \ldots\}$ converging to a real number x is: $\forall \varepsilon > 0, \exists N$ such that $\forall i > N, |x_i - x| < \varepsilon$. In other words, we can make the difference between x and the terms of the sequence as small as possible by looking sufficiently far out in the sequence. The nonstandard

definition of convergence is that $\forall$ infinite i, $|x_i - x|$ is infinitesimal. (We can take "infinite" to mean "$1/i$ is infinitesimal"). The nonstandard definition has many fewer quantifiers than the standard definition. Also, it is more intuitive (x_i for "large" i are "close to" x.). In fact, this is the major reason for formulating our Theory in terms of nonstandard mathematics: all the definitions become simpler when formulated in nonstandard terms.

In this section we give nonstandard equivalents of asymptotic satisfaction for standard programs P and standard specifications R. The nonstandard versions are actually logically equivalent to the standard ones in IST. Because IST is conservative over ZFC, any statement in the language of ordinary mathematics (e.g. statements about error magnitudes) which we prove using nonstandard methods and the nonstandard definition of asymptotic satisfaction will be provable using standard methods and the standard definition. In general, however, the nonstandard proofs are much more intuitive and much easier to construct and read.

For the remainder of the discussion we fix a standard program P and a standard specification R for P.

Definition 3.3.1: If T is a model for P, T is *hyperaccurate* iff T satisfies all standard asymptotic axioms, i.e. iff $\forall^{st} V, \varepsilon, \delta, \alpha[T \in \mathsf{startup}(V, \delta) \wedge T \in \mathsf{primacc}_\alpha(V, \varepsilon, \delta)]$

□

Definition 3.3.2: If T is a model for P, T *hypersatisfies* R iff T satisfies all standard asymptotic specifications for R, i.e. iff $\forall^{st} V, \varepsilon, \delta[T \in \mathsf{startup}(V, \delta) \wedge T \in \mathsf{accuracy}(V, \varepsilon, \delta)]$.

□

In IST we have the following equivalence: P asymptotically satisfies R iff every hyperaccurate model of P hypersatisfies R.

We will next obtain more useful characterizations of a model being hyperaccurate and a model hypersatisfying a specification.

Definition 3.3.3: If $x, y \in \mathbf{R}$, $x \approx y$ (read "x is infinitely close to y") iff $|x - y|$ is infinitesimal. If V, V' are variable assignments, $V \approx V'$ iff $\rho(V, V')$ is infinitesimal.

□

$V \approx V'$ iff V and V' make the same variables defined and undefined, and assign the same values to the integer variables, and for all real variables X, $V(\mathrm{X}) \approx V'(\mathrm{X})$.

In IST we have the following equivalence: a model T of P satisfies all standard **startup** axioms iff $\forall^{\mathrm{st}} V, \exists \alpha, V'[\langle\langle \alpha, V' \rangle\rangle \in T \wedge V' \approx V]$. In particular, if T is hyperaccurate then any standard V can be approximated infinitely closely by a V' that T can start up with.

In IST we have the following equivalence: a model T of P satisfies all standard $\mathbf{accuracy}_R$ axioms iff $\forall^{\mathrm{st}} V \in \mathrm{dom}(R), \alpha, V'$, if $\langle\langle \alpha, V' \rangle\rangle \in T$ and $V' \approx V$ then:

1. There are no infinite paths through T whose first element is $\langle \alpha, V' \rangle$.
2. For every σ maximal in T, if σ's first element is $\langle \alpha, V' \rangle$ then the last element of σ is $\langle \beta, U \rangle$ and $\exists W[R(V, W) \wedge W \approx U]$.

In particular, if T hypersatisfies R then if we start up T with an infinitely close approximation to $V \in \mathrm{dom}(R)$, T will eventually terminate with a variable assignment which is infinitely close to some assignment W such that $R(V, W)$.

Definition 3.3.4: A real number x is *finite* iff there exists a standard y such that $|x| < y$. An integer is finite iff the corresponding real is finite. A variable assignment V is finite iff every variable V assigns is assigned a finite value (whether integer or real); equivalently, iff $\exists^{\mathrm{st}} V'$ such that $V \approx V'$.

□

In IST we have the following equivalence: if α is an assignment node with

label $v := F(v_1, \ldots, v_l)$ and $F \neq$ "/", then a model T of P satisfies all $\mathsf{primacc}_\alpha$ axioms iff $\forall \sigma, e, e'$, if:

- $\sigma \hat{} \langle e, e' \rangle \in T$
- $e = \langle \alpha, V \rangle$
- V is finite
- V assigns values to $v_1, \ldots, v_l$ and $F(V(v_1), \ldots, V(v_l))\downarrow$

then $e' \neq !$ and e' assigns v a value w such that $w \approx F(V(v_1), \ldots, V(v_l))$.

For division, we have the following equivalence: If α is an assignment node with label $a := b/c$ then a model T of P satisfies all standard $\mathsf{primacc}_\alpha$ axioms iff $\forall \sigma, e, e'$, if:

- $\sigma \hat{} \langle e, e' \rangle \in T$
- $e = \langle \alpha, V \rangle$
- V is finite
- V assigns values to b and c and $V(c)$ is not infinitesimal

then $e' \neq !$ and e' assigns a a value w such that $w \approx V(b)/V(c)$.

In particular, if T is hyperaccurate then computations of operations other than division on finite inputs introduce only infinitesimal error, and computations of division on finite inputs only introduce infinitesimal error when not dividing by an infinitesimal.

By the above facts, if we want to prove that a standard program asymptotically satisfies a standard relation R, it is sufficient to let T be an arbitrary hyperaccurate model, and prove that it hypersatisfies R.

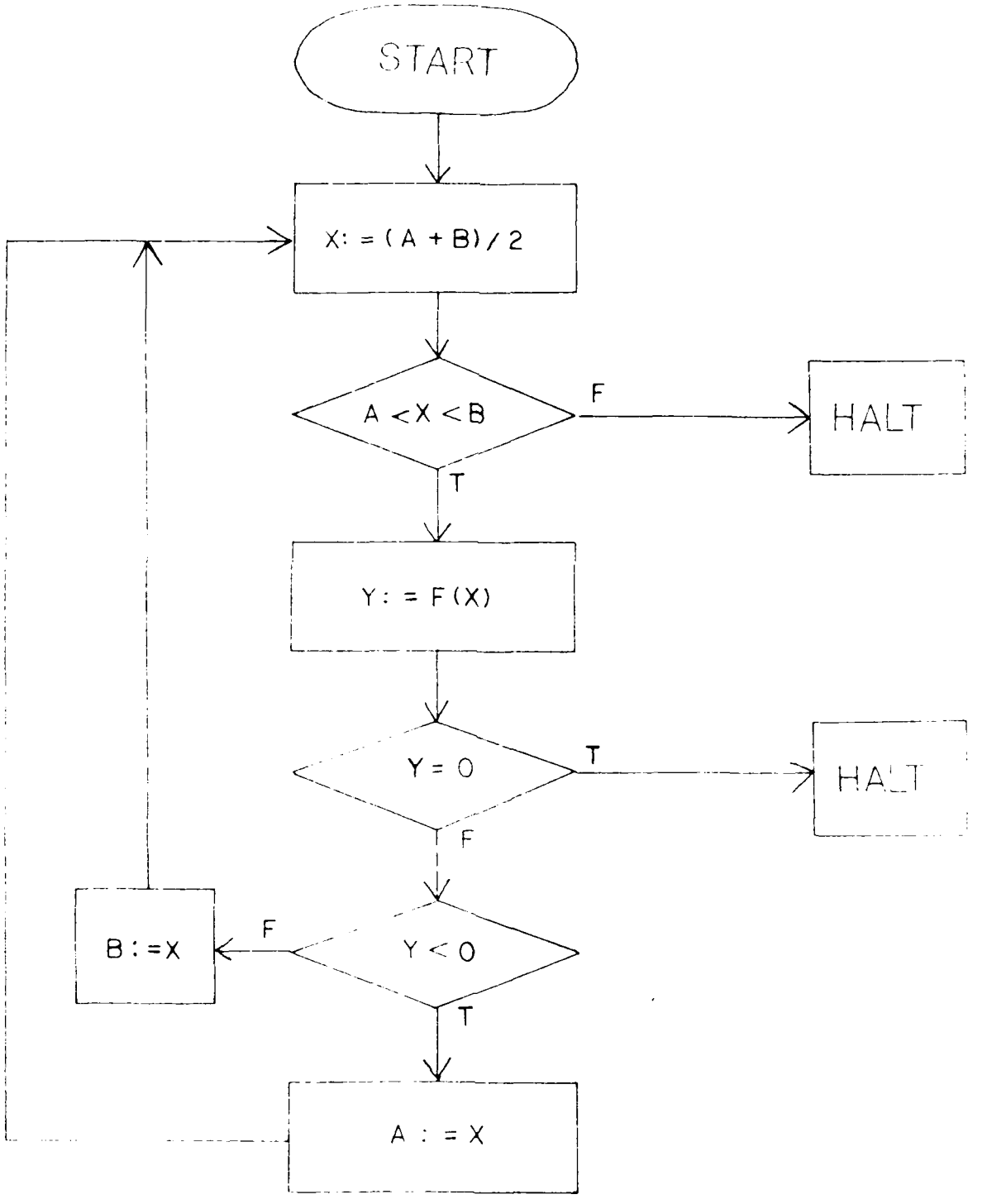
Chapter 4

A Sample Verification

In this chapter we apply the Theory to verify (informally) the asymptotic correctness of a program P to find roots of a standard continuous function $f : \mathbf{R} \to \mathbf{R}$. We will freely use elementary facts from nonstandard analysis without proving them; the details can be found in, e.g. [1]. In particular, we will need to use the nonstandard definition of continuity of f in the verification. In nonstandard analysis, a standard function is continuous if, for every standard x, if $y \approx x$ then $f(y) \approx f(x)$.

The flow chart for the program is pictured on the next page. It has 3 real-valued variables, A, B, X and Y. What we will verify about the program is that it asymptotically satisfies the following condition: if it is started up with A and B defined and A $<$ B and $f(\mathrm{A}) < 0 < f(\mathrm{B})$ then it will eventually terminate with X defined and $f(\mathrm{X}) = 0$. We know from the Intermediate Value Theorem of real analysis that such a root must exist.

Let's recall what it means for a program to meet such a specification asymptotically. It means that if we have fixed numbers x and y, and $x < y$ and $f(x) < 0 < f(y)$, and a $\varepsilon > 0$, then on a sufficiently large machine, if we run the program with A and B sufficiently close to x and y, then the program will terminate with a value for X that is within ε of a root of f. We will have verified this statement if, assuming the program is started up with A and B infinitely close to standard x and y such that $x < y$ and


START
X: = (A + B) / 2
A < X < B
F
HALT
T
Y: = F(X)
Y = 0
T
HALT
F
B: = X
F
Y < 0
T
A : = X

$f(x) < 0 < f(y)$, and assuming that the program's arithmetic operations only introduce infinitesimal error on finite values, we can prove that the program terminates with X defined and infinitely close to a standard root z of f. (We must also assume that f is computed with only infinitesimal error on finite elements. This would presumably be done by some other program which P would call which had been verified to compute f asymptotically. We will assume for simplicity that there is some function f_m such that the machine computed value of $f(x)$ is $f_m(x)$. In general, of course, P need not compute the same value for $f(x)$ twice in a row. The program can be verified without this assumption, but the proof in that case involves details which would be counterproductive here).

The program attempts to find a root by the method of bisection. It executes a loop in which, in each pass through, it does the following: it first takes the midpoint of its current 2 endpoints, and computes the value of f there. If it is 0, the program halts. If it is negative, the midpoint becomes the "new" lower endpoint, and the loop continues. If it is positive, the midpoint becomes the "new" upper endpoint, and the loop continues.

How do we make sure that the program terminates? If it were running on a machine with ideal arithmetic, it would be entirely possible that the program would never actually find a root, but would just get values of A and B that were closer and closer to a root. We know this can't happen on a finite machine, however, because to do so would require that A and B pass through an infinite number of distinct real values in the course of running the program. What *would* happen on a finite machine. On a very accurate but finite machine, execution would look very much like execution on an ideal machine for a while. As the values of A and B got very close to each other, however, there would come a point where the distance between A and B was less than the roundoff error in computing the midpoint of the two. This would result in the program computing a value for the midpoint which would round to one of the endpoints, or possibly even to a number outside the endpoints. Since boolean tests are exact, we can detect this condition P check after each computation of the midpoint to see if the computed value is between the endpoints; if it is not, the program terminates.

The argument we have just given proves that the program always terminates

normally *if* there are no unhandled exceptions. Notice that there are no "exception" arrows in our flow chart, so we had better be able to prove that no unhandled exceptions occur. It's easy to show that there are no exception due to referencing undefined variables, since we assume that A and B are defined initially and every other variable is assigned to before the first time it is referenced. The only other kinds of exception that can occur are exceptions due to attempting to evaluate an expression on arguments that are not finite, and attempting to divide by an infinitesimal. The latter kind can't happen because the only division in the program is division by 2, which is not infinitesimal. To show that the former sort can't happen it would suffice to show that whenever control reachs an assignment statement, the values of A, B and X are finite (when defined), since these are the only variables which appear on the right hand side of an assignment statement. We will argue something stronger, namely that whenever control reachs an assignment statement, the values of A, B and X (when defined) are all between the initial two values of A and B. Call these initial values a and b respectively. We prove this statement by induction on the number of steps the program has executed. Suppose that there is some integer n such that after n steps, control comes to an assignment statement and one of A, B or X is defined and not between a and b. Choose n as small as possible. We consider each assignment statement separately, and show for each one that control cannot be at the statement at time n.

X := (A+B)/2. The first time control reachs this statement X is undefined and $A = a$ and $B = b$. Thus, n cannot correspond to the first time control reachs this point. Any other time control reachs this point, it must have been at B := X or A := X after $n - 1$ steps. By minimality of n, A, B and X must all have been between a and b at step $n - 1$, and since at step $n - 1$ we are only assigning one variable the value of another, the values of the three variables must be between a and b after executing step $n - 1$ and so also before executing step n.

Y := *f*(X). If control is at this statement at time n then it was at X := (A + B)/2 at time $n - 2$. By minimality of n, this means that A and B must have been between a and b before executing step $n - 2$, and therefore after, since step $n - 2$ only assigns to X. At step $n - 1$ control must have been at the test statement A < X < B. If control passed to Y := f(X) rather

than HALT, it must be that the value of X at time $n-1$ was between the values of A and B, and therefore between a and b. Test do not affect the values of variables, so all three variables would have to have been between a and b at time n.

A := X. If control is at this statement at time n then it must have been at statement Y := f(X) at time $n-3$. By minimality of n, the values of A, B and X must have been between a and b at time $n-3$, and none of the statements executed at times $n-3, n-2$ and $n-1$ affect the values of A, B or X, so they must still be between a and b at time n.

B := X. The argument here is identical to that for the previous case.

This establishes that no exception occurs in the program, so it terminates normally. It obviously terminates with X defined, because this happens at the first assignment statement. It is also easy to prove by induction that at all points in the execution of P, A < B and the values of A and B are such that the *machine-computed* value of f(A) is < 0 and the *machine-computed* value of f(B) is < 0. This is ensured by the Y = 0 and Y < 0 test. We emphasize that the *actual* values may not have the same sign as the machine-computed values, but we don't need them to be the same sign to verify our program. We will now prove directly that X is infinitely close to a root of f at termination. There are two cases, corresponding to the two HALT statements.

If P halts after the A < X < B test, we claim it must be the case that A, B and X are all infinitely close to each other. Prior to the test, X was assigned to (A + B)/2. The computation of this expression can introduce infinitesimally much error, so all we really know is that after the assignment statement,

$$X \approx \frac{A+B}{2}$$

Since control passes to HALT after the test, it must be the case that either X $\leq$ A or X $\leq$ B. Consider the first case. Since A < B,

$$A < \frac{A+B}{2}$$

Therefore, A is between X and $(A + B)/2$, and the last two numbers are infinitely close. A, X and $(A + B)/2$ are all infinitely close to each other. Also, this means that

$$\frac{A+B}{2} - A = \frac{B-A}{2}$$

is infinitesimal, so $B - A$ is infinitesimal, so $A \approx B$. By elementary nonstandard analysis, all three of A, B and X must therefore be close to a single standard real number z. Also, by the assumption that P asymptotically computes f and f is continuous,

$$\begin{aligned} f_n(A) &\approx f(A) \\ &\approx f(z) \\ &\approx f(B) \\ &\approx f_m(B) \end{aligned}$$

But the first and last numbers are of opposite sign. The only way two numbers can be infinitely close to each other and of opposite sign is if they are infinitesimal. Therefore, $f(z)$ is infinitesimal. But f and z are standard, so $f(z)$ is standard, and the only standard number which is infinitesimal is 0. Therefore, z is a standard real root of f, and the program terminates with $X \approx z$.

Suppose P halts after the $Y = 0$ test. There exists some standard z infinitely close to X, so

$$\begin{aligned} 0 &= f_n(X) \\ &\approx f(X) \\ &\approx f(z) \end{aligned}$$

so again, we have $f(z)$ infinitesimal, which implies that z must be a standard root of f, and the program terminates with $X \approx z$.

Appendix A

Notation

In this Appendix we list some notations that we've used in the preceding chapters.

- $x \in X$ means "x is an element of (set) X." $X \subseteq Y$ means "X is a subset of Y."
- $\langle x_1, \ldots, x_n \rangle$ is the finite sequence with entries $x_1, \ldots, x_n$ (in that order). $\langle\rangle$ is the unique sequence of length 0, or the empty sequence.
- $\sigma \preceq \tau$ means the sequence τ extends the sequence σ to the right.
- $\sigma^\frown\tau$ stands for the concatenation of the sequences σ and τ.
- $|x|$ is the absolute value of x.
- $f : D \to R$ means "f is a function from D into R."
- $t\downarrow$ means "t is defined". $t\uparrow$ means "t is undefined." $s \simeq t$ mean "s is defined iff t is, and if s and t are defined, they are equal."
- $\forall^{\mathrm{st}} x.\, \phi$ means "for all standard x, ϕ holds." $\exists^{\mathrm{st}} x.\, \phi$ means "there exists standard x such that ϕ holds."

Bibliography

[1] A. Hurd and P.A. Loeb (1985). *Introduction to Nonstandard Real Analysis*. Academic Press, New York.

[2] E. Nelson (1977). Internal Set Theory. *Bull. Amer. Math. Soc.* **83**.

DISTRIBUTION LIST

adcresses	number of copies
Donald M. Elefante RADC/COTC	7
RADC/DOVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	12
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
Director DMAAC (Attn: RE) 3200 S. Second St. St Louis MO 63118-3399	1
AFCSA/SAMI Attn: Miss Griffin 10363 Pentagon Wash DC 20330-5425	1

HQ USAF/SCTT
Pentagon
Wash DC 20330-5190 1

SAF/AQSC
Pentagon 4D-267
Wash DC 20330-1000 1

DIRECTOR
DMAHTC
ATTN: SDSIM
Wash DC 20315-0030 1

Director, Info Systems
OASD (C3I)
Rm 3E187
Pentagn
Wash DC 20301-3040 1

Fleet Analysis Center
Attn: GIDEP Operations Center
Code 30G1 (E. Richards)
Corona CA 91720 1

HQ AFSC/DLAE
ANDREWS AFB DC 20334-5000 1

HQ AFSC/XRT
Andrews AFB MD 20334-5000 1

HQ AFSC/XRK
ANDREWS AFB MD 20334-500 1

HQ SAC/NRI
OFFUTT AFB NE 68113-5001 1

HQ SAC/SCPT
OFFUTT AFB NE 68113-5001 1

HQ ESD/DOOA
Attn: Fred Ladwig
San Antonio TX 78243-5000 1

DTESA/RQEE
ATTN: LARRY G.MCMANUS
2501 YALE STREET SE
Airport Plaza, Suite 102
ALBUQUERQUE NM 87106 1

HQ TAC/DRIY
Attn: Mr. Westerman
Langley AFB VA 23665-5001 1

HQ TAC/DOA
LANGLEY AFB VA 23665-5001 1

HQ TAC/DRCC
LANGLEY AFB VA 23665-5001 1

HQ TAC/DRCA
LANGLEY AFB VA 23665-5001 1

HQ AFOTEC (OAWD) 1
Attn: Capt. Novack)
KIRTLAND AFB NM 87117-7001

ASD/ENEMS 2
Wright-Patterson AFB OH 45433-6503

ASD-AFALC/AXP 1
WRIGHT-PATTERSON AFB OH 45433

ASD/AFALC/AXAE 1
Attn: W. H. Dungey
Wriight-Patterson AFB OH 45433-6533

ASD/ENAMW 1
Wright-Patterson AFB OH 45433-6503

ASD/ENAMA 1
Wright-Patterson AFB OH 45433

AFIT/LDEE 1
BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

AFWAL/MLPO 1
Attn: G. H. Griffith
Wright-Patterson AFB OH 45433-6533

AFWAL/MLPO
WRIGHT-PATTERSON AFB OH 45433-6533 1

AFWAL/MLTE
WRIGHT-PATTERSON AFB OH 45433 1

AFWAL/FIES/SURVIAC
WRIGHT-PATTERSON AFB OH 45433 1

AAMRL/HE
WRIGHT-PATTERSON AFB OH 45433-6573 1

Air Force Human Resources Laboratory
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433 1

2750 ABW/SSLT
Bldg 262
Post 11S
Wright-Patterson AFB OH 454433 1

AFHRL/OTS
WILLIAMS AFB AZ 85240-6457 1

1843EIG/EIEM
HICKAM AFB HI 96854 1

AUL/LSE 1
MAXWELL AFB AL 36112-5564

HQ AFSPACECOM/XPYS 1
ATTN: DR. WILLIAM R. MATOUSH
PETERSON AFB CO 80914-5001

3280TTG/RISS 1
Attn: TSgt Kirk
Lackland AFB TX 78236

HQ Air Training Command 1
TTOI
Randolph AFB TX 78150-5001

HQ ATC/TTOK 1
Randolph AFB TX 78150-5001

Defense Communications Engineering Ctr 1
Technical Library
1860 Wiehle Avenue
Reston VA 22090-5500

COMMAND CONTROL AND COMMUNICATIONS DIV 1
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION COMMAND
ATTN: CODE DICA
QUANTICO VA 22134-5080

AFLMC/LGY 1
ATTN: CH, SYS ENGR DIV
GUNTER AFS AL 36114

U.S. Army Strategic Defense Command 1
Attn: DASD-H-MPL
P.O. Box 1500
Huntsville AL 35807-3801

COMMANDING OFFICER 1
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46219-2189

COMMANDING OFFICER 1
NAVAL TRAINING SYSTEMS CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100

COMMANDER 1
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 96428
SAN DIEGO CA 92152-5000

COMMANDER (CODE 3433) 1
ATTN: TECHNICAL LIBRARY
NAVAL WEAPONS CENTER
CHINA LAKE, CALIFORNIA 93555-6001

SUPERINTENDENT (CODE 1424) 1
NAVLA POST GRADUATE SCHOOL
MONTEREY CA 93943-5000

COMMANDING OFFICER 2
NAVAL RESEARCH LABORATORY
ATTN: CODE 2627
WASHINGTON DC 20375-5000

SPACE & NAVAL WARFARE SYSTEMS COMMAND 1
PMW 153-3DP
ATTN: R. SAVARESE
WASHINGTON DC 20363-5100

CDR, U.S. ARMY MISSILE COMMAND 2
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN: AMSMI-RD-CS-R (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5241

Advisory Group on Electron Devices 2
Hammond John/Technical Info Coordinator
201 Varick Street, Suite 1140
New York NY 10014

UNIVERSITY OF CALIFORNIA/LOS ALAMOS 1
NATIONAL LABORATORY
ATTN: DAN BACA/REPORT LIBRARIAN
P.O. BOX 1663, MS-P364
LOS ALAMOS NM 87545

RAND CORPORATION THE/LIBRARY 1
HELFER DORIS S/HEAD TECH SVCS
P.O. BOX 2138
SANTA MONICA CA 90406-2138

AEDC LIBRARY (TECH REPORTS FILE) 1
MS-100
ARNOLD AFS TN 37389-9998

USAG 1
Attn: ASH-PCA-CRT
Ft Huachuca AZ 85613-6000

DOT LIBRARY/10A SECTION 1
ATTN: M493.2
800 INDEPENDENCE AVE. S.W.
WASH DC 20591

1839 EIG/EIET (KENNETH W. IRBY) 1
KEESLER AFB MS 39534-6348

JTFPMO
Attn: Technical Director
1500 Planning Research Drive
McLean VA 22102

2

AWS TECHNICAL LIBRARY
FL4414
SCOTT AFB IL 62225-5458

1

HQ ESC/CWPP
San Antonio TX 78243-5000

1

AFEWC/ESRI
SAN ANTONIO TX 78243-5000

4

485 FIG/FIER (DMO)
GRIFFISS AFB NY 13441-4348

2

ESD/XRS
ATTN: ADV SYS DEV
HANSCOM AFB MA 01731-5000

1

ESD/ICP
HANSCOM AFB MA 01731-5000

1

ESD/XRSE
BLDG 1704
HANSCOM AFB MA 01731-5000

2

HQ ESD SYS-2
HANSCOM AFB MA 01731-5000

1

ESD/TCD-2 1
ATTN: CAPTAIN J. MEYER
HANSCOM AFB MA 01731-5000

The Software Engineering Institute 1
Attn: Major Dan Burton, USAF
580 South Aiken Avenue
Pittsburgh PA 15232-1502

DIRECTOR 1
NSA/CSS
ATTN: T513/TDL (DAVID MARJARUM)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: W166
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R-8316 (MR. ALLEY)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R24
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R21
9800 SAVAGE ROAD
FORT GEORGE G MEASDE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: DEFSMAC (JAMES E. HILLMAN)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R31
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R5
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R8
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R9
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: S031
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: S21
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: V307
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: W07
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR
NSA/CSS
ATTN: W3
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: R523
FORT GEORGE G MEADE MD 20755-6000 2

DIRECTOR
NSA/CSS
ATTN: R53 (JOHN C. DAVIS)
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000 1

DoD COMPUTER SECURITY CENTER
ATTN: C4/TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000 1

Odyssey Research Associates
310A Harris B. Dates Drive
Ithaca, NY 14850-1313 5

SDIO/S-BM
ATTN: Lt Col Sowa
The Pentagon
Washington, DC 20301-7100 1

SDIO/S-BM
ATTN: Capt Hart
The Pentagon
Washington, DC 20301-7100 1

SDIO/S-BM
ATTN: Cdr Newton
The Pentagon
Washington, DC 20301-7100 1

SDIO/S-BM 1
ATTN: Lt Col Rindt
The Pentagon
Washington, DC 20301-7100

SDIO Library 1
IDA 1801 N. Beauregard Street
Alexandria, VA 22311

SAF/AQSD 1
ATTN: Lt Col Ben Greenway
The Pentagon
Washington, DC 20330

AFSC/CV-D 1
ATTN: Lt Col Flynn
Andrews AFB, MD 20334-5000

HQ SD/XR 1
ATTN: Col Heimach
P.O. Box 92960
Worldway Postal Center
Los Angeles, CA 90009-2960

SD/CN 1
ATTN: Col Wilkenson
P.O. Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

SD/CNI 1
ATTN: Col Hohman
P.O. Box 92960
Worldway Postal Center
Los Angeles, CA 90009-2960

SD/CNIS 1
ATTN: Lt Col Pennell
P.O. Box 2960
Worldway Postal Center
Los Angeles, CA 90009-2960

SD/CNW
P.O. Box 2960
Worldway Postal Center
Los Angeles, CA 90009-2960 1

SD/CWX
P.O. Box 2960
Worldway Postal Center
Los Angeles, CA 90009-2960 1

SD/CNB
P.O. Box 2960
Worldway Postal Center
Los Angeles, CA 90009-2960 1

ESD/AT
ATTN: Col Paul
Hanscom AFB, MA 01731-5000 1

ESD/ATS
ATTN: Lt Col Oldenberg
Hanscom AFB, MA 01731-5000 1

ESD/ATN
ATTN: Lt Col Leib
Hanscom AFB, MA 01731-5000 1

AFSTC/XLX
ATTN: Lt Col Detucci
Kirtland AFB, NM 87117 1

USA SDC/CASD-H-SR
ATTN: Larry Tubbs
P.O. Box 1500
Huntsville, AL 35807 1

ANSER Corp
Suite 800
Crystal Gateway 3
1215 Jefferson Davis Highway
Arlington, VA 22202

IDA
ATTN: Albert Perrella
1801 N. Beauregard Street
Alexandria, VA 22311

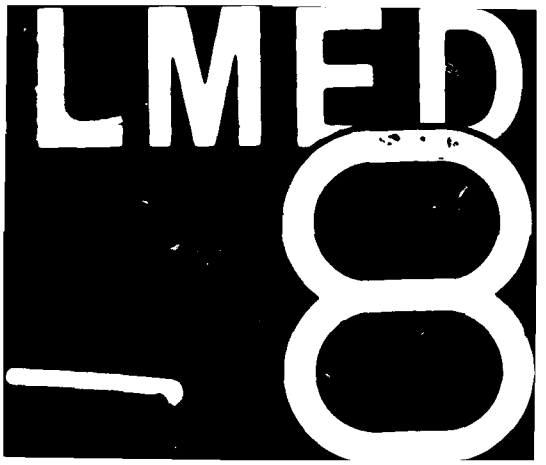
AFOTEC/XPP
ATTN: Capt Wrotel
Kirtland AFB, NM 87117

AF Space Command/XPXIS
Peterson AFB, CO 80914-5001

Director NSA
ATTN: George Hoover, V43
9800 Savage Road
Ft George G. Meade, MD 20755-6000

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.


LMED
8